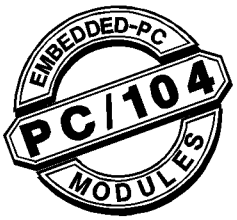


DPM104HR Dual Port SRAM Interface Board User's Manual

Real Time Devices Finland Oy

HARDWARE RELEASE 2.1

Real Time Devices Finland Oy
Lepolantie 14
FIN-00660 Helsinki, Finland
Tel: (+358) 9 346 4538
Fax: (+358) 9 346 4539
Email: sales@rtdfinland.fi
URL: www.rtdfinland.fi

(c) RTD Finland Oy 1997-2001

IMPORTANT

Although the information contained in this manual has been carefully verified, RTD Finland Oy assumes no responsibility for errors that might appear in this manual, or for any damage to things or persons resulting from improper use of this manual or from the related software. RTD Finland Oy reserves the right to change the contents of this manual, as well as the features and specifications of this product at any time, without notice.

Published by
Real Time Devices Finland Oy
Lepolantie 14
FIN-00660 Helsinki, Finland

Copyright © 1997-2001 by RTD Finland Oy
All rights reserved

Printed in Finland

Table of Contents Page

INTRODUCTION
- **Dual Port Memory**
- **Mechanical description**
- **Connector description**
- **What comes with your board**
- **Board accessories**
  - **Application software and drivers**
  - **Hardware accessories**
- **Using this manual**
- **When you need help**

CHAPTER 1 - BOARD SETTINGS
- **Factory configured jumper settings**
- **Base address jumpers**
- **Interrupt channels**

CHAPTER 2 - BOARD INSTALLATION
- **Board installation**

CHAPTER 3 - HARDWARE DESCRIPTION
- **Dual Port Memory**
- **Interrupts**
- **Semaphores**
- **Backup Battery connection**

CHAPTER 4 - BOARD OPERATION AND PROGRAMMING
- **Defining the memory map**
- **DPM datasheet reprint from IDT**
- **Interrupts**
  - **What is an interrupt?**
  - **Interrupt request lines**
  - **8259 Programmable interrupt controller**
  - **Interrupt mask register (IMR)**
  - **End-Of-Interrupt (EOI) Command**
  - **What exactly happens when an interrupt occurs?**
  - **Using interrupts in your program**
  - **Writing an interrupt service routine (ISR)**
  - **Saving the startup IMR and interrupt vector**
  - **Common Interrupt mistakes**

APPENDIX A - DPM104HR Specifications

List of Illustrations

Page

To be completed later

(c) RTD Finland Oy 1997-2001

INTRODUCTION

 (c) RTD Finland Oy 1997-2001

This user's manual describes the operation of the DPM104HR Dual Port Memory interface board.

Some of the key properties of the DPM104HR include:

- 8K x 16 True Dual Ported memory cells with simultaneous reads from the same memory location
- Mapped into 18KB of host low memory
- On-chip arbitration logic
- Full on-chip hardware and software for semaphore signalling between ports
- Full asyncronous operation from either port
- Interrupts from both sides
- Backup with external battery and power supply supervisor
- +5V only operation
- -40 to +85C Operating temperature range
- PC/104 compliant

The following paragraphs briefly describe the major features of the DPM104HR. A more detailed discussion is included in Chapter 3 (Hardware description) and in Chapter 4 (Board operation and programming. The board setup is described in Chapter 1 (Board Settings). A full description of the Dual Port Memory chip is included in Chapter 4.

Dual Port Memory

The DPM104HR dual port memory interface is implemented using monolithic memory chip. This chip provides a high speed low power asyncronous acess to a total of 16 KB of RAM memory. Inter-port arbitratration is integrated into the chip. To enable high speed data transfers hardware semaphores and interrupts are supported.

To maintain memory contents in cases of power loss an external battery may be used to provide power for the memory.

Mechanical description

The DPM104HR is designed on a PC/104 form factor. An easy mechanical interface to PC/104 systems can be achieved. Stack two PC/104 compliant cpuModules directly on your DPM104HR using the onboard mounting holes.

Connector description

There are two 16-bit PC/104 bus connectors on the DPM104HR to directly interface to two cpuModules with 16-bit busses. One connector on the top side of the board is the Master side of the DPM and the other connector on the bottom of the board is the Slave side.

 (c) RTD Finland Oy 1997-2001

Note! Only 16-bit CPU bus boards may be used with the DPM104HR.

What comes with your board

You receive the following items in your DPM104HR package:

* DPM104HR Dual Port Memory interface module
* User's manual

Note: DOS/WIN95/98/2000/NT 4.0 drivers and test software are available from our website at www.rtdfinland.fi.

If any item is missing or damaged, please call Real Time Devices Finland customer service department at (+358) 9 346 4538.

Board accessories

In addition to the items included in your DPM104HR delivery several software and hardware accessories are available. Call your distributor for more information on these accessories and for help in choosing the best items to support your distributed control system.

Using this manual

This manual is intended to help you install your new DPM104HR card and get it running quickly, while also providing enough detail about the board and it's functions so that you can enjoy maximum use of it's features even in the most demanding applications.

When you need help

This manual and all the example programs will provide you with enough information to fully utilize all the features on this board. If you have any problems installing or using this board, contact our Technical Support Department (+358) 9 346 4538 during European business hours, or send a FAX to (+358) 9 346 4539 or Email to sales@rtdfinland.fi. When sending a FAX or Email request, please include your company's name and address, your name, your telephone number, and a brief description of the problem.

 (c) RTD Finland Oy 1997-2001

CHAPTER 1 - BOARD SETTINGS

The DPM104HR board has jumper settings you can change to suit your application and host computer memory configuration. The factory settings are listed and shown in the diagram in the beginning of this chapter.

 (c) RTD Finland Oy 1997-2001

Factory configured Jumper Settings

Table 1-1 illustrates the factory jumper setting for the DPM104HR. Figure 1-1 shows the board layout of the DPM104HR and the locations of the jumpers. The following paragraphs explain how to change the factory jumper settings to suit your specific application.

Table 1-1 Factory configured jumper settings see figure 1-1 for detailed locations

JUMPER NAME	DESCRIPTION OF JUMPER	NUMBER OF JUMPERS	FACTORY SETTING JUMPERS INSTALLED
ADDR_A	BASE ADDRESS	5	D8000
ADDR_B	BASE ADDRESS	5	D8000
IRQ_A	HOST INTERRUPT	5	Not connected
IRQ_B	HOST INTERRUPT	5	Not connected

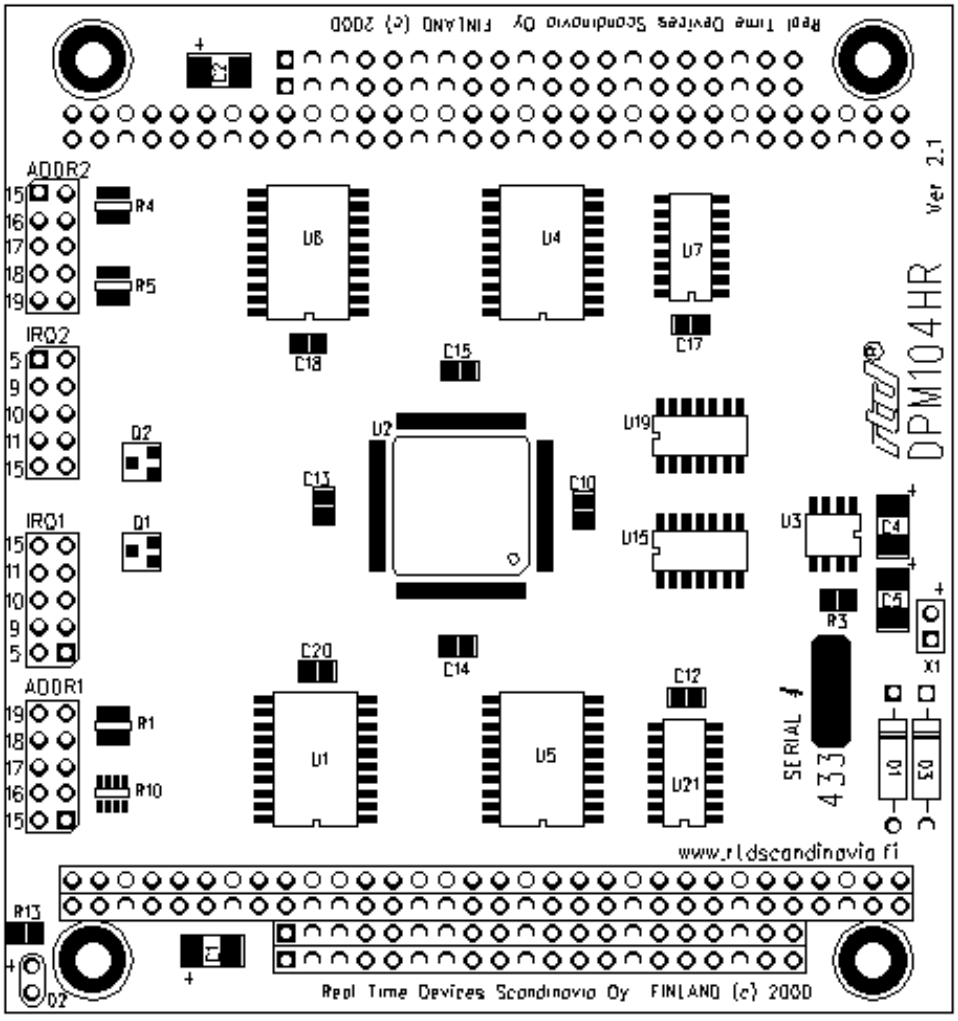


Fig. 1-1 - Board layout showing jumper locations

Base Address Jumpers (Factory setting: D8000h)

The DPM104HR is memory mapped into the low memory of your host computer. The board occupies a memory window of 18KB starting from the base address.

The most common cause of failure when you are first setting up your module is address contention. Some of your computers memory space is already occupied by other devices and memory resident programs. When the DPM104HR attempts to use it's reserved memory addresses already used by another device, erratic performance may occur and data read from the board may be corrupted.

To avoid this problem make sure you set up the base address first using the 5 jumpers marked "ADDR" which let you choose from number of addresses in your host computers memory map. Should the factory installed setting of D8000h be unusable for your system configuration, you may change this setting to another using the options illustrated in Table 1-2 and in Figures 1-2 and 1-3 . The table shows the jumper settings and their corresponding values in hexadecimal values. Make sure you verify the correct location of the base address jumpers. When the jumper is removed it corresponds to a logical "1", connecting the jumper to a "0". When you set the base address of the module, record the setting in the table inside the back cover of this manual after the Appendices.

Note: If you are using a memory manager such as QEMM, make sure you exclude the memory section you are occupying by the DPM104HR; for example X=D000-D8FF.

 (c) RTD Finland Oy 1997-2001

Address Settings for DPM104HR	
Base Address Hex	Jumper Settings 18 17 16 1
80XXX	0 0 0 0
88XXX	0 0 0 1
90XXX	0 0 1 0
98XXX	0 0 1 1
A0XXX	0 1 0 0
A8XXX	0 1 0 1
B0XXX	0 1 1 0
B8XXX	0 1 1 1
C0XXX	1 0 0 0
C8XXX	1 0 0 1
D0XXX	1 0 1 0
D8XXX	1 0 1 1
E0XXX	1 1 0 0
E8XXX	1 1 0 1
F0XXX	1 1 1 0
F8XXX	1 1 1 1
1 = NO JUMPER, 0 = JUMPER installed Note : A19 is always decoded as 1 !	

Table 1-2 Base address jumper settings

Note: The above table illustrates only the settings for the address bits A18-A15.

Address line A19 should always be decoded as "1" since low memory area 00000 -7ffff is normally occupied by the system.

(c) RTD Finland Oy 1997-2001

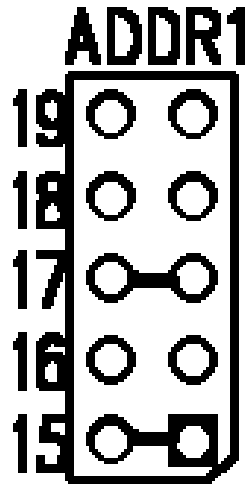


Fig. 1-2 Base address jumpers for side A (Master)

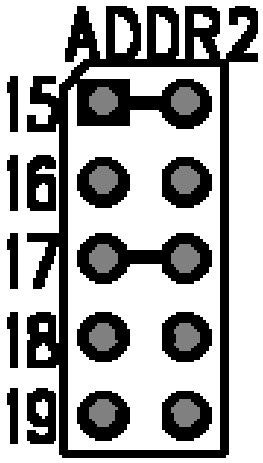


Fig. 1-3 Base address jumpers for side B (Slave)

(c) RTD Finland Oy 1997-2001

Interrupt Channel A (Factory setting: Not connected)

The header connector, shown on Figure 1-4a, lets you connect the onboard DPM master side interrupt output to one of the interrupt channels available on the host AT bus.

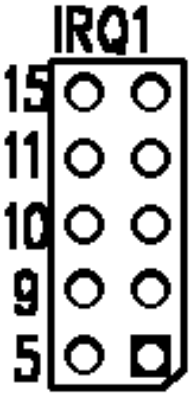


Fig. 1-4a Interrupt jumpers for side A

The header connector, shown on Figure 1-4b, lets you connect the onboard DPM slave side interrupt output to one of the interrupt channels available on the host AT bus.

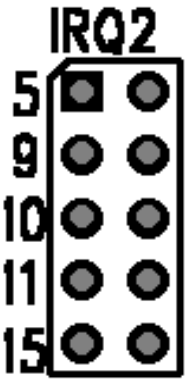


Fig. 1-4b Interrupt jumpers for side B

 (c) RTD Finland Oy 1997-2001

CHAPTER 2 - BOARD INSTALLATION

The DPM104HR memory interface board is very easy to connect to your industrial distributed control system. Direct interface two PC/104 computers in one stack! This chapter tells you step-by-step how to install your DPM104HR into your system.

Board Installation

Keep your board in its antistatic bag until you are ready to install it to your system! When removing it from the bag, hold the board at the edges and do not touch the components or connectors. Please handle the board in an antistatic environment and use a **grounded** workbench for testing and handling of your hardware.

Before installing the board in your computer, check the jumper settings. Chapter 1 reviews the factory settings and how to change them. If you need to change any settings, refer to the appropriate instructions in Chapter 1. Note that incompatible jumper settings can result in unpredictable board operation and erratic response.

General installation guidelines:

1. Turn OFF the power to your computer and all devices connected to DPM104HR
2. Touch the grounded metal housing of your computer to discharge any antistatic buildup and then remove the board from its antistatic bag.
3. Hold the board by it's edges and install it in an enclosure or place it on the table on an antistatic surface.
4. Connect the board to the two PC/104 cpuModules using the two bus interface connectors.

Installation integrated with a PC/104 module stack:

* Secure the two PC/104 installation holes opposite to the bus connectors with standoffs.

* Connect the DPM104HR board to the two PC/104 cpuModules using the two bus interface connectors.

 (c) RTD Finland Oy 1997-2001

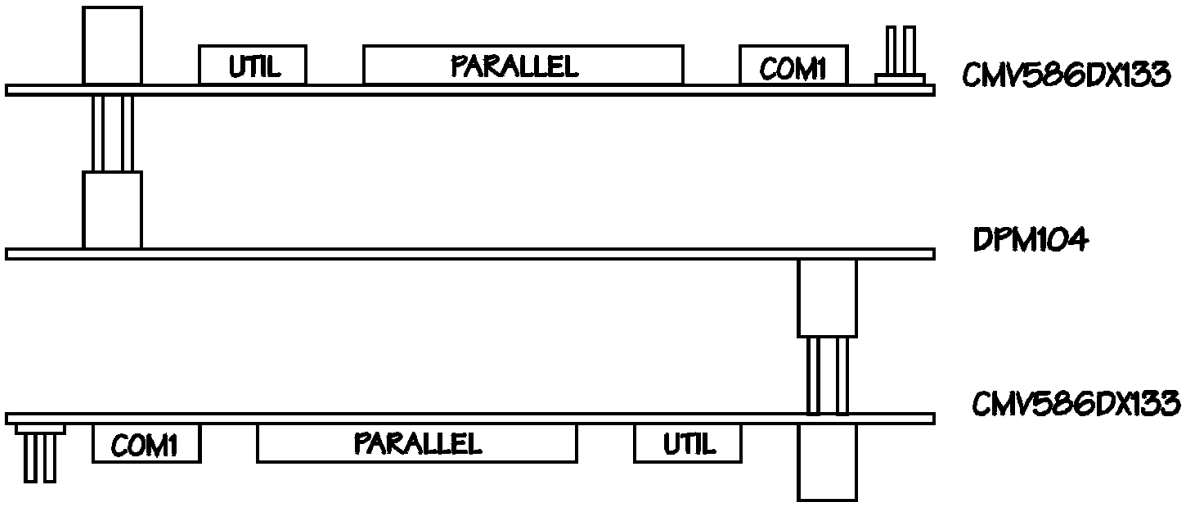


Fig. 2-1 DPM104HR integrated with two PC/104 cpuModule stacks

Note: The default connectors on the DPM104HR are two soldertail PC/104 bus connectors. Other connector options are available upon request.

CHAPTER 3 - HARDWARE DESCRIPTION

Chapter 3 - **Hardware Description** describes the major features of the DPM104HR: the Dual Port Memory chip, Interrupts , Semaphores and Backup battery supply.

Figure 3-1 shows the general block diagram of the DPM104HR. This chapter describes the major features of the DPM104HR: the Dual Port Memory chip, Interrupts and the Battery supply.

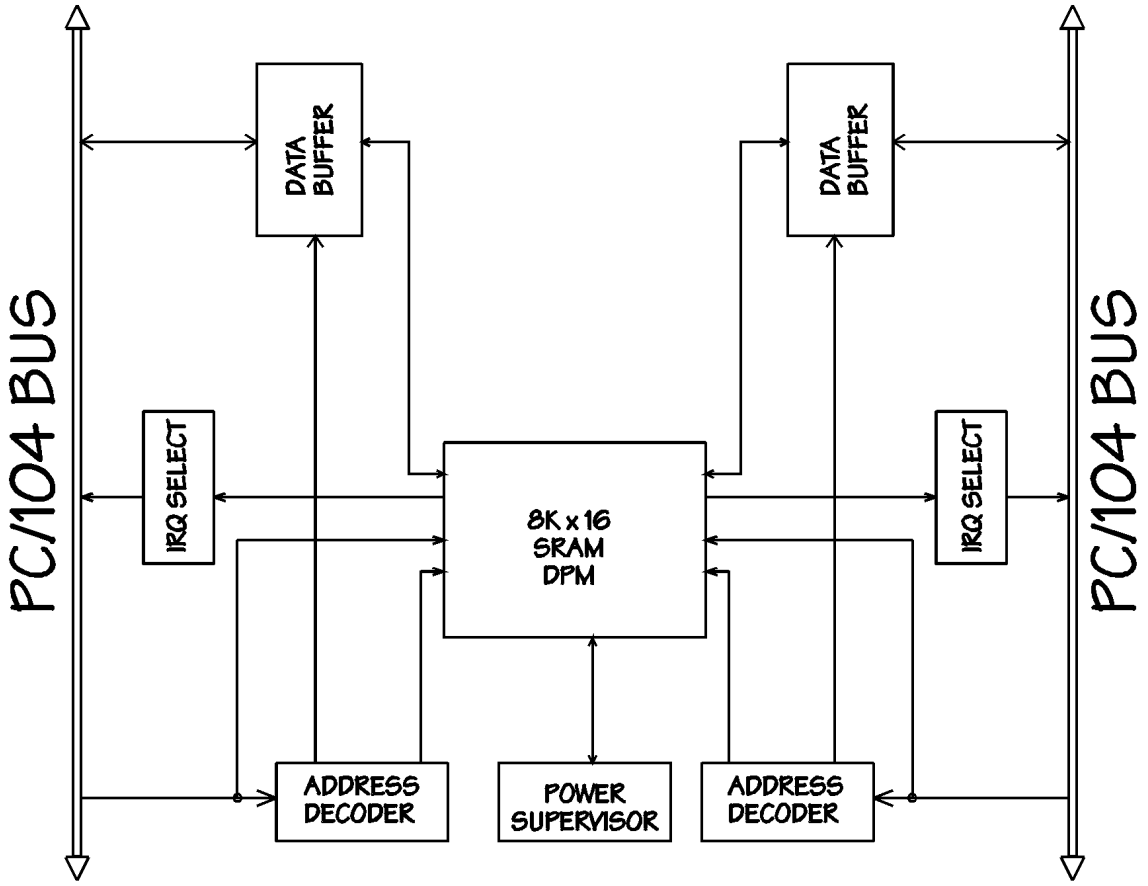


Fig. 3-1 DPM104HR Block diagram

 (c) RTD Finland Oy 1997-2001

Dual Port Memory chip

The onboard memory is a 8K x 16 Dual port static RAM. The device provides two independent ports with separate control, access and I/O pins that permit independent, asyncronous access for reads and writes to any location in the memory. Automatic powerdown in activated when the chip is not addressed. Interrupts and semaphores are supported, the onchip busy flag is not used.

Interrupts

If you wish to use the interrupt supported by the Dual port memory chips, a memory mailbox is assigned to each port. The Slave port interrupt flag is set when the Master port writes to memory location FFE (hex). The Slave port clears the interrupt by reading the same address. Likewise for the Master writing to address FFF (hex). The messages at addresses FFE and FFF (hex) are user definable. Refer to component specific datasheet for more detailed information on the functionality of interrupts.

Semaphores

The Dual port RAM chip has eight locations dedicated to binary semaphore flags. These flags allow the CPU on the Slave or Master side to claim a priviledge over the other CPU for functions defined by the system software. As an example, the semaphore can be used by one CPU to inhibit the other CPU from accessing blocks of memory or shared resources.

Software handshaking between CPU's offers the maximum in system flexibility by permitting shared resources to be allocated in various system configurations. The DPM chip does not use the semaphore to control any resources through hardware, thus allowing the system designer maximum flexibility. Refer to component specific datasheet for more detailed information on how to use semaphores.

Battery supply

The DPM104HR board has an onboard power supply supervisor to monitor the RAM supply voltage. A small 10 micro Farad charge capacitor will provide enough power for small transient power losses. This capacitor is kept in charge while power is maintained normally. For completely non volatile operation an external 3,6V battery must be used in header connector X1 located next to the Master 16-bit PC/104 bus connector. The polarity of the battery is important, see picture below for correct connection of the external battery.

Pin 1 - GND (square pad)
Pin 2 - 3,6V battery power

 (c) RTD Finland Oy 1997-2001

CHAPTER 4 - BOARD OPERATION AND PROGRAMMING

This chapter shows you how to program and use your DPM104HR. It provides a complete detailed description of the memory map and a detailed discussion of programming operations to aid you in programming. The full functionality of the Dual Port Memory chip is described in the datasheet reprint from IDT. You may use the diagnostics and software supplied by RTD to fully test your system under different operating systems. Please download the latest drivers and software form our website: <www.rtdfinland.fi>.

Defining the Memory Map

The memory map of the DPM memory occupies 18 Kbytes of host CPU memory space. This window is freely selectable by the user as described in Chapter 1, Table 1-2. After setting the base address you have access to the internal resources of the DPM-chip as described in the next sections reprinted from the IDT chip datasheet.

The memory map of the DPM chip resources is illustrated in the table below.

Address Range in HEX	Memory Resources
000-1FFE	Dual Port Memory mailbox area
1FFE	Interrupt clear for Master (Read)
1FFE	Interrupt to Master (Write)
1FFF	Interrupt clear for Slave (Read)
1FFF	Interrupt to Slave (Write)
2000-2008	Semaphore bits for both sides

 (c) RTD Finland Oy 1997-2001

Integrated Device Technology, Inc.

HIGH-SPEED 8K x 16 DUAL-PORT STATIC RAM

IDT7025S/L

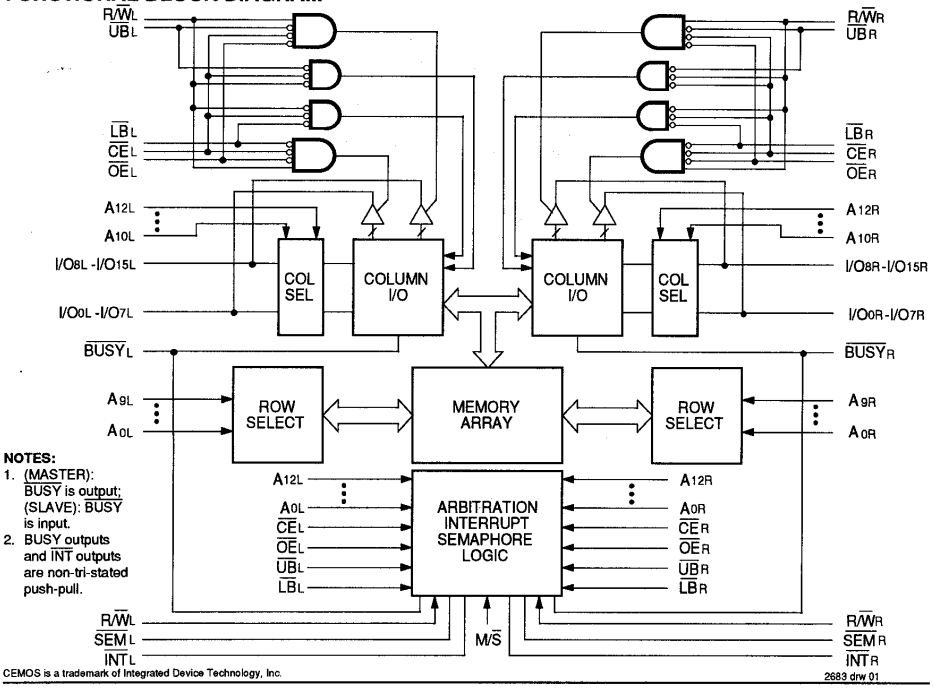
FEATURES:

- True dual-ported memory cells which allow simultaneous reads of the same memory location
- High-speed access
  - — Military: 35/45/55/70ns (max.)
  - — Commercial: 25/30/35/45/55ns (max.)
- Low-power operation
  - — IDT7025S
    Active: 750mW (typ.)
    Standby: 5mW (typ.)
  - — IDT7025L
    Active: 750mW (typ.)
    Standby: 1mW (typ.)
- Separate upper-byte and lower-byte control for multiplexed bus compatibility
- IDT7025 easily expands data bus width to 32 bits or more using the Master/Slave select when cascading more than one device
- M/$\overline{S}$ = H for $\overline{BUSY}$ output flag on Master
  M/$\overline{S}$ = L for $\overline{BUSY}$ input on Slave
- Interrupt Flag
- On-chip port arbitration logic
- Full on-chip hardware support of semaphore signaling between ports
- Fully asynchronous operation from either port
- Battery backup operation—2V data retention
- TTL compatible, single 5V (±10%) power supply
- Available in 84-pin PGA, quad flatpack and PLCC
- Industrial temperature range (-40°C to +85°C) is available, tested to military electrical specifications

DESCRIPTION:

The IDT7025 is a high-speed 8K x 16 dual-port static RAM. The IDT7025 is designed to be used as a stand-alone 128K-

FUNCTIONAL BLOCK DIAGRAM



NOTES:
1. (MASTER): $\overline{BUSY}$ is output; (SLAVE): $\overline{BUSY}$ is input.
2. BUSY outputs and $\overline{INT}$ outputs are non-tri-stated push-pull.

CEMOS is a trademark of Integrated Device Technology, Inc.

2683 drw 01

©1992 Integrated Device Technology, Inc.

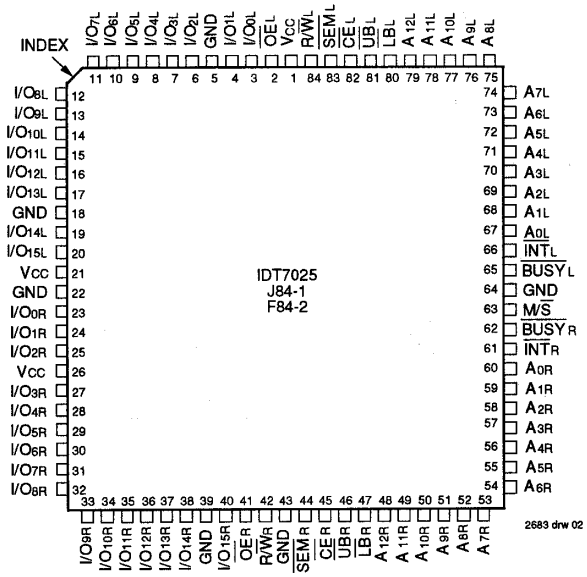
bit dual-port RAM or as a combination MASTER/SLAVE dual-port RAM for 32-bit-or-more word systems. Using the IDT MASTER/SLAVE dual-port RAM approach in 32-bit or wider memory system applications results in full-speed, error-free operation without the need for additional discrete logic.

This device provides two independent ports with separate control, address, and I/O pins that permit independent, asynchronous access for reads or writes to any location in memory. An automatic power down feature controlled by $\overline{CE}$ permits the on-chip circuitry of each port to enter a very low standby power mode.

Fabricated using IDT's CEMOS™ high-performance technology, these devices typically operate on only 750mW of power at maximum access times as fast as 25ns. Low-power (L) versions offer battery backup data retention capability with typical power consumption of 500µW from a 2V battery.

The IDT7025 is packaged in a ceramic 84-pin PGA, an 84-pin quad flatpack, and a PLCC. The military devices are processed 100% in compliance to the test methods of MIL-STD-883, Method 5004.

PIN CONFIGURATIONS



PLCC/FLATPACK
TOP VIEW

NOTES:
1. All Vcc pins must be connected to power supply.
2. All GND pins must be connected to ground supply.

6

(c) RTD Finland Oy 1997-2001

TRUTH TABLE: NON-CONTENTION READ/WRITE CONTROL

Inputs[1]						Outputs		Mode
$\overline{CE}$	R/$\overline{W}$	$\overline{OE}$	$\overline{UB}$	$\overline{LB}$	$\overline{SEM}$	I/O8-15	I/O0-7	
H	X	X	X	X	H	Hi-Z	Hi-Z	Deselected: Power Down
X	X	X	H	H	H	Hi-Z	Hi-Z	Both Bytes Deselected: Power Down
L	L	X	L	H	H	DATAIN	Hi-Z	Write to Upper Byte Only
L	L	X	H	L	H	Hi-Z	DATAIN	Write to Lower Byte Only
L	L	X	L	L	H	DATAIN	DATAIN	Write to Both Bytes
L	H	L	L	H	H	DATAOUT	Hi-Z	Read Upper Byte Only
L	H	L	H	L	H	Hi-Z	DATAOUT	Read Lower Byte Only
L	H	L	L	L	H	DATAOUT	DATAOUT	Read Both Bytes
X	X	H	X	X	X	Hi-Z	Hi-Z	Outputs Disabled

2683 tbl 01

NOTE:
1. $A_{0L} — A_{12L} \neq A_{0R} — A_{12R}$

TRUTH TABLE: SEMAPHORE READ/WRITE CONTROL

Inputs						Outputs		Mode
$\overline{CE}$	R/$\overline{W}$	$\overline{OE}$	$\overline{UB}$	$\overline{LB}$	$\overline{SEM}$	I/O8-15	I/O0-7	
H	H	L	X	X	L	DATAOUT	DATAOUT	Read Data in Semaphore Flag
X	H	L	H	H	L	DATAOUT	DATAOUT	Read Data in Semaphore Flag
H	↑	X	X	X	L	DATAIN	DATAIN	Write DIN0 into Semaphore Flag
X	↑	X	H	H	L	DATAIN	DATAIN	Write DIN0 into Semaphore Flag
L	X	X	L	X	L	—	—	Not Allowed
L	X	X	X	L	L	—	—	Not Allowed

2683 tbl 02

ABSOLUTE MAXIMUM RATINGS[1]

Symbol	Rating	Commercial	Military	Unit
V_{TERM}[2]	Terminal Voltage with Respect to GND	-0.5 to +7.0	-0.5 to +7.0	V
T_A	Operating Temperature	0 to +70	-55 to +125	°C
T_{BIAS}	Temperature Under Bias	-55 to +125	-65 to +135	°C
T_{STG}	Storage Temperature	-55 to +125	-65 to +150	°C
I_{OUT}	DC Output Current	50	50	mA

2683 tbl 04

NOTE:
1. Stresses greater than those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect reliability.
2. V_{TERM} must not exceed V_{CC} + 0.5V.

RECOMMENDED OPERATING TEMPERATURE AND SUPPLY VOLTAGE

Grade	Ambient Temperature	GND	Vcc
Military	-55°C to +125°C	0V	5.0V ± 10%
Commercial	0°C to +70°C	0V	5.0V ± 10%

2683 tbl 05

RECOMMENDED DC OPERATING CONDITIONS

Symbol	Parameter	Min.	Typ.	Max.	Unit
V_{CC}	Supply Voltage	4.5	5.0	5.5	V
GND	Supply Voltage	0	0	0	V
V_{IH}	Input High Voltage	2.2	—	6.0[2]	V
V_{IL}	Input Low Voltage	-0.5[1]	—	0.8	V

2683 tbl 06

NOTE:
1. $V_{IL} \geq$ -3.0V for pulse width less than 20ns.
2. V_{TERM} must not exceed V_{CC} + 0.5V.

CAPACITANCE (T_A = +25°C, f = 1.0MHz)

Symbol	Parameter[1]	Conditions	Max.	Unit
C_{IN}	Input Capacitance	V_{IN} = 0V	11	pF
C_{OUT}	Output Capacitance	V_{OUT} = 0V	11	pF

2682 tbl 03

NOTE:
1. This parameter is determined by device characterization but is not production tested.

DPM104HR (c) RTD Finland Oy 1997-2001

6

TRUTH TABLES

TRUTH TABLE I — INTERRUPT FLAG(1)

Left Port					Right Port					Function
$R/\overline{W}_L$	$\overline{CE}_L$	$\overline{OE}_L$	A_{0L}-A_{12L}	$\overline{INT}_L$	$R/\overline{W}_R$	$\overline{CE}_R$	$\overline{OE}_R$	A_{0R}-A_{12R}	$\overline{INT}_R$	
L	L	X	1FFF	X	X	X	X	X	L(2)	Set Right $\overline{INT}_R$ Flag
X	X	X	X	X	X	L	L	1FFF	H(3)	Reset Right $\overline{INT}_R$ Flag
X	X	X	X	L(3)	L	L	X	1FFE	X	Set Left $\overline{INT}_L$ Flag
X	L	L	1FFE	H(2)	X	X	X	X	X	Reset Left $\overline{INT}_L$ Flag

NOTES:
1. Assumes $\overline{BUSY}_L = \overline{BUSY}_R$ = H.
2. If $\overline{BUSY}_L$ = L, then no change.
3. If $\overline{BUSY}_R$ = L, then no change.

2683 tbl 15

DPM104HR (c) RTD Finland Oy 1997-2001

TRUTH TABLE II — ADDRESS BUSY ARBITRATION

Inputs			Outputs		
$\overline{CE}_L$	$\overline{CE}_R$	A_{0L}-A_{12L} A_{0R}-A_{12R}	$\overline{BUSY}_L$(1)	$\overline{BUSY}_R$(1)	Function
X	X	NO MATCH	H	H	Normal
H	X	MATCH	H	H	Normal
X	H	MATCH	H	H	Normal
L	L	MATCH	(2)	(2)	Write Inhibit(3)

2683 tbl 16

NOTES:
1. Pins $\overline{BUSY}_L$ and $\overline{BUSY}_R$ are both outputs when the part is configured as a master. Both are inputs when configured as a slave. $\overline{BUSY}$x outputs on the IDT7025 are push pull, not open drain outputs. On slaves the $\overline{BUSY}$x input internally inhibits writes.
2. L if the inputs to the opposite port were stable prior to the address and enable inputs of this port. H if the inputs to the opposite port became stable after the address and enable inputs of this port. If t_{APS} is not met, either $\overline{BUSY}_L$ or $\overline{BUSY}_R$ = Low will result. $\overline{BUSY}_L$ and $\overline{BUSY}_R$ outputs cannot be low simultaneouly.
3. Writes to the left port are internally ignored when $\overline{BUSY}_L$ outputs are driving low regardless of actual logic level on the pin. Writes to the right port are internally ignored when $\overline{BUSY}_R$ outputs are driving low regardless of actual logic level on the pin.

TRUTH TABLE III — EXAMPLE OF SEMAPHORE PROCUREMENT SEQUENCE(1)

Functions	D_0 - D_{15} Left	D_0 - D_{15} Right	Status
No Action	1	1	Semaphore free
Left Port Writes "0" to Semaphore	0	1	Left port has semaphore token
Right Port Writes "0" to Semaphore	0	1	No change. Right side has no write access to semaphore
Left Port Writes "1" to Semaphore	1	0	Right port obtains semaphore token
Left Port Writes "0" to Semaphore	1	0	No change. Left port has no write access to semaphore
Right Port Writes "1" to Semaphore	0	1	Left port obtains semaphore token
Left Port Writes "1" to Semaphore	1	1	Semaphore free
Right Port Writes "0" to Semaphore	1	0	Right port has semaphore token
Right Port Writes "1" to Semaphore	1	1	Semaphore free
Left Port Writes "0" to Semaphore	0	1	Right port has semaphore token
Left Port Writes "1" to Semaphore	1	1	Semaphore free

2683 tbl 17

NOTE:
1. This table denotes a sequence of events for only one of the eight semaphores on the IDT7025.

FUNCTIONAL DESCRIPTION

The IDT7025 provides two ports with separate control, address and I/O pins that permit independent access for reads or writes to any location in memory. The IDT7025 has an automatic power down feature controlled by $\overline{CE}$. The $\overline{CE}$ controls on-chip power down circuitry that permits the respective port to go into a standby mode when not selected ($\overline{CE}$ high). When a port is enabled, access to the entire memory array is permitted.

INTERRUPTS

If the user chooses to use the interrupt function, a memory location (mail box or message center) is assigned to each port. The left port interrupt flag ($\overline{INT}_L$) is set when the right port writes to memory location 1FFE (HEX). The left port clears the interrupt by reading address location 1FFE. Likewise, the right port interrupt flag ($\overline{INT}_R$) is set when the left port writes to memory location 1FFF (HEX) and to clear the interrupt flag ($\overline{INT}_R$), the right port must read the memory location 1FFF. The message (16 bits) at 1FFE or 1FFF is user-defined. If the interrupt function is not used, address locations 1FFE and 1FFF are not used as mail boxes, but as part of the random access memory. Refer to Table I for the interrupt operation.

BUSY LOGIC

Busy Logic provides a hardware indication that both ports of the RAM have accessed the same location at the same time. It also allows one of the two accesses to proceed and signals the other side that the RAM is "busy". The busy pin can then be used to stall the access until the operation on the other side is completed. If a write operation has been attempted from the side that receives a busy indication, the write signal is gated internally to prevent the write from proceeding.

The use of busy logic is not required or desirable for all applications. In some cases it may be useful to logically OR the busy outputs together and use any busy indication as an interrupt source to flag the event of an illegal or illogical

DPM104HR (c) RTD Finland Oy 1997-2001

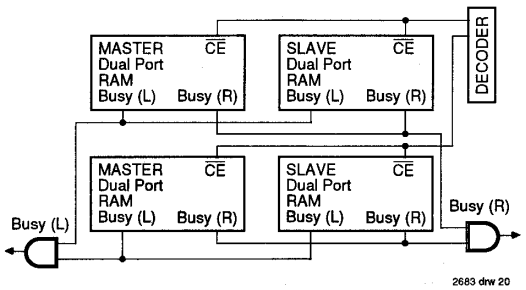


Figure 3. Busy and chip enable routing for both width and depth expansion with IDT7025 RAMs.

operation. If the write inhibit function of busy logic is not desirable, the busy logic can be disabled by placing the part in slave mode with the M/$\overline{S}$ pin. Once in slave mode the $\overline{BUSY}$ pin operates solely as a write inhibit input pin. Normal operation can be programmed by tying the $\overline{BUSY}$ pins high. If desired, unintended write operations can be prevented to a port by tying the busy pin for that port low.

The busy outputs on the IDT 7025 RAM in master mode, are push-pull type outputs and do not require pull up resistors to operate. If these RAMs are being expanded in depth, then the busy indication for the resulting array requires the use of an external AND gate.

WIDTH EXPANSION WITH BUSY LOGIC MASTER/SLAVE ARRAYS

When expanding an IDT7025 RAM array in width while using busy logic, one master part is used to decide which side of the RAM array will receive a busy indication, and to output that indication. Any number of slaves to be addressed in the same address range as the master, use the busy signal as a write inhibit signal. Thus on the IDT7025 RAM the busy pin is an output if the part is used as a master (M/$\overline{S}$ pin = H), and the busy pin is an input if the part used as a slave (M/$\overline{S}$ pin = L) as shown in Figure 3.

If two or more master parts were used when expanding in width, a split decision could result with one master indicating busy on one side of the array and another master indicating busy on one other side of the array. This would inhibit the write operations from one port for part of a word and inhibit the write operations from the other port for the other part of the word.

The busy arbitration, on a master, is based on the chip enable and address signals only. It ignores whether an access is a read or write. In a master/slave array, both address and chip enable must be valid long enough for a busy flag to be output from the master before the actual write pulse can be initiated with either the R/$\overline{W}$ signal or the byte enables. Failure to observe this timing can result in a glitched internal write inhibit signal and corrupted data in the slave.

SEMAPHORES

The IDT7025 is an extremely fast dual-port 8K x 16 CMOS static RAM with an additional 8 address locations dedicated to binary semaphore flags. These flags allow either processor on the left or right side of the dual-port RAM to claim a privilege over the other processor for functions defined by the system designer's software. As an example, the semaphore can be used by one processor to inhibit the other from accessing a portion of the dual-port RAM or any other shared resource.

The dual-port RAM features a fast access time, and both ports are completely independent of each other. This means that the activity on the left port in no way slows the access time of the right port. Both ports are identical in function to standard CMOS static RAM and can be read from, or written to, at the same time with the only possible conflict arising from the simultaneous writing of, or a simultaneous READ/WRITE of, a non-semaphore location. Semaphores are protected against such ambiguous situations and may be used by the system program to avoid any conflicts in the non-semaphore portion of the dual-port RAM. These devices have an automatic power-down feature controlled by $\overline{CE}$, the dual-port RAM enable, and $\overline{SEM}$, the semaphore enable. The $\overline{CE}$ and $\overline{SEM}$ pins control on-chip power down circuitry that permits the respective port to go into standby mode when not selected. This is the condition which is shown in Truth Table where $\overline{CE}$ and $\overline{SEM}$ are both high.

Systems which can best use the IDT7025 contain multiple processors or controllers and are typically very high-speed systems which are software controlled or software intensive. These systems can benefit from a performance increase offered by the IDT7025's hardware semaphores, which provide a lockout mechanism without requiring complex programming.

Software handshaking between processors offers the maximum in system flexibility by permitting shared resources to be allocated in varying configurations. The IDT7025 does not use its semaphore flags to control any resources through hardware, thus allowing the system designer total flexibility in system architecture.

An advantage of using semaphores rather than the more

DPM104HR (c) RTD Finland Oy 1997-2001

common methods of hardware arbitration is that wait states are never incurred in either processor. This can prove to be a major advantage in very high-speed systems.

HOW THE SEMAPHORE FLAGS WORK

The semaphore logic is a set of eight latches which are independent of the dual-port RAM. These latches can be used to pass a flag, or token, from one port to the other to indicate that a shared resource is in use. The semaphores provide a hardware assist for a use assignment method called "Token Passing Allocation." In this method, the state of a semaphore latch is used as a token indicating that shared resource is in use. If the left processor wants to use this resource, it requests the token by setting the latch. This processor then verifies its success in setting the latch by reading it. If it was successful, it proceeds to assume control over the shared resource. If it was not successful in setting the latch, it determines that the right side processor has set the latch first, has the token and is using the shared resource. The left processor can then either repeatedly request that semaphore's status or remove its request for that semaphore to perform another task and occasionally attempt again to gain control of the token via the set and test sequence. Once the right side has relinquished the token, the left side should succeed in gaining control.

The semaphore flags are active low. A token is requested by writing a zero into a semaphore latch and is released when the same side writes a one to that latch.

The eight semaphore flags reside within the IDT7025 in a separate memory space from the dual-port RAM. This address space is accessed by placing a low input on the $\overline{SEM}$ pin (which acts as a chip select for the semaphore flags) and using the other control pins (Address, $\overline{OE}$, and R/$\overline{W}$) as they would be used in accessing a standard static RAM. Each of the flags has a unique address which can be accessed by either side through address pins A0 – A2. When accessing the semaphores, none of the other address pins has any effect.

When writing to a semaphore, only data pin Do is used. If a low level is written into an unused semaphore location, that flag will be set to a zero on that side and a one on the other side (see Table III). That semaphore can now only be modified by the side showing the zero. When a one is written into the same location from the same side, the flag will be set to a one for both sides (unless a semaphore request from the other side is pending) and then can be written to by both sides. The fact that the side which is able to write a zero into a semaphore subsequently locks out writes from the other side is what makes semaphore flags useful in interprocessor communications. (A thorough discussing on the use of this feature follows shortly.) A zero written into the same location from the other side will be stored in the semaphore request latch for that side until the semaphore is freed by the first side.

When a semaphore flag is read, its value is spread into all data bits so that a flag that is a one reads as a one in all data bits and a flag containing a zero reads as all zeros. The read value is latched into one side's output register when that side's semaphore select ($\overline{SEM}$) and output enable ($\overline{OE}$) signals go active. This serves to disallow the semaphore from changing state in the middle of a read cycle due to a write cycle from the other side. Because of this latch, a repeated read of a semaphore in a test loop must cause either signal ($\overline{SEM}$ or $\overline{OE}$) to go inactive or the output will never change.

A sequence WRITE/READ must be used by the semaphore in order to guarantee that no system level contention will occur. A processor requests access to shared resources by attempting to write a zero into a semaphore location. If the semaphore is already in use, the semaphore request latch will contain a zero, yet the semaphore flag will appear as one, a fact which the processor will verify by the subsequent read (see Table III). As an example, assume a processor writes a zero to the left port at a free semaphore location. On a subsequent read, the processor will verify that it has written successfully to that location and will assume control over the resource in question. Meanwhile, if a processor on the right side attempts to write a zero to the same semaphore flag it will fail, as will be verified by the fact that a one will be read from that semaphore on the right side during subsequent read. Had a sequence of READ/WRITE been used instead, system contention problems could have occurred during the gap between the read and write cycles.

It is important to note that a failed semaphore request must be followed by either repeated reads or by writing a one into the same location. The reason for this is easily understood by looking at the simple logic diagram of the semaphore flag in Figure 4. Two semaphore request latches feed into a semaphore flag. Whichever latch is first to present a zero to the semaphore flag will force its side of the semaphore flag low and the other side high. This condition will continue until a one is written to the same semaphore request latch. Should the other side's semaphore request latch have been written to a zero in the meantime, the semaphore flag will flip over to the other side as soon as a one is written into the first side's request latch. The second side's flag will now stay low until its semaphore request latch is written to a one. From this it is easy to understand that, if a semaphore is requested and the processor which requested it no longer needs the resource, the entire system can hang up until a one is written into that semaphore request latch.

The critical case of semaphore timing is when both sides request a single token by attempting to write a zero into it at the same time. The semaphore logic is specially designed to resolve this problem. If simultaneous requests are made, the logic guarantees that only one side receives the token. If one side is earlier than the other in making the request, the first side to make the request will receive the token. If both requests arrive at the same time, the assignment will be arbitrarily made to one port or the other.

One caution that should be noted when using semaphores is that semaphores alone do not guarantee that access to a resource is secure. As with any powerful programming technique, if semaphores are misused or misinterpreted, a software error can easily happen.

Initialization of the semaphores is not automatic and must be handled via the initialization program at power-up. Since any semaphore request flag which contains a zero must be reset to a one, all semaphores on both sides should have a

 (c) RTD Finland Oy 1997-2001

one written into them at initialization from both sides to assure that they will be free when needed.

USING SEMAPHORES—SOME EXAMPLES

Perhaps the simplest application of semaphores is their application as resource markers for the IDT7025's dual-port RAM. Say the 8K x 16 RAM was to be divided into two 4K x 16 blocks which were to be dedicated at any one time to servicing either the left or right port. Semaphore 0 could be used to indicate the side which would control the lower section of memory, and Semaphore 1 could be defined as the indicator for the upper section of memory.

To take a resource, in this example the lower 4K of dual-port RAM, the processor on the left port could write and then read a zero in to Semaphore 0. If this task were successfully completed (a zero was read back rather than a one), the left processor would assume control of the lower 4K. Meanwhile the right processor was attempting to gain control of the resource after the left processor, it would read back a one in response to the zero it had attempted to write into Semaphore 0. At this point, the software could choose to try and gain control of the second 4K section by writing, then reading a zero into Semaphore 1. If it succeeded in gaining control, it would lock out the left side.

Once the left side was finished with its task, it would write a one to Semaphore 0 and may then try to gain access to Semaphore 1. If Semaphore 1 was still occupied by the right side, the left side could undo its semaphore request and perform other tasks until it was able to write, then read a zero into Semaphore 1. If the right processor performs a similar task with Semaphore 0, this protocol would allow the two processors to swap 4K blocks of dual-port RAM with each other.

The blocks do not have to be any particular size and can even be variable, depending upon the complexity of the software using the semaphore flags. All eight semaphores could be used to divide the dual-port RAM or other shared resources into eight parts. Semaphores can even be assigned different meanings on different sides rather than being given a common meaning as was shown in the example above.

Semaphores are a useful form of arbitration in systems like disk interfaces where the CPU must be locked out of a section of memory during a transfer and the I/O device cannot tolerate any wait states. With the use of semaphores, once the two devices has determined which memory area was "off-limits" to the CPU, both the CPU and the I/O devices could access their assigned portions of memory continuously without any wait states.

Semaphores are also useful in applications where no memory "WAIT" state is available on one or both sides. Once a semaphore handshake has been performed, both processors can access their assigned RAM segments at full speed.

Another application is in the area of complex data structures. In this case, block arbitration is very important. For this application one processor may be responsible for building and updating a data structure. The other processor then reads and interprets that data structure. If the interpreting processor reads an incomplete data structure, a major error condition may exist. Therefore, some sort of arbitration must be used between the two different processors. The building processor arbitrates for the block, locks it and then is able to go in and update the data structure. When the update is completed, the data structure block is released. This allows the interpreting processor to come back and read the complete data structure, thereby guaranteeing a consistent data structure.

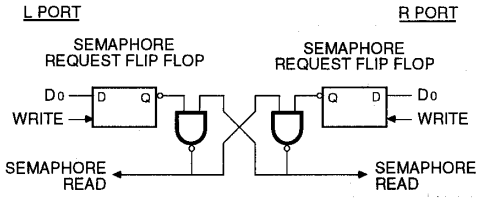


Figure 4. IDT7025 Semaphore Logic

 (c) RTD Finland Oy 1997-2001

Interrupts

- What is an interrupt ?

An interrupt is an event that causes the processor in your computer to temporarily halt its current process and execute another routine. Upon completion of the new routine, control is returned to the original routine at the point where its execution was interrupted.

Interrupts are a very flexible way of dealing with asynchronous events. Keyboard activity is a good example; your computer cannot predict when you might press a key and it would be a waste of processor time to do nothing while waiting for a keystroke to occur. Thus the interrupt scheme is used and the processor proceeds with other tasks. Then when a keystroke occurs, the keyboard 'interrupts' the processor , and the processor gets the keyboard data, placed it into memory, and then returns to what it was doing before the interrupt occurred. Other common devices that use interrupts are network boards, A/D boards, serial ports etc.

You can interrupt the other processor on the other side of the DPM by writing to the topmost addresses of the memory at addresses 1FFE and 1FFF. When an interrupt is received you may signal that data has been updated in the memory. By using interrupts you can write powerful code to interface to your DPM104HR.

-Interrupt request lines

To allow different peripheral devices to generate interrupts on the same computer, the PC AT bus has interrupt request channels (IRQ's). A rising edge transition on one of these lines will be latched into the interrupt controller. The interrupt controller checks to see if the interrupts are to be acknowledged from that IRQ and, if another interrupt is being processed, it decides if the new request should supersede the one in progress or if it has to wait until the one in progress is done. The priority level of the interrupt is determined by the number of the IRQ; IRQ0 has the highest priority IRQ15 the lowest. Many of the IRQ's are used by the standard system resources. IRQ0 is dedicated for the internal timer, IRQ1 is dedicated to the keyboard input, IRQ3 for serial port COM2 and IRQ4 for serial port COM1. Often interrupts 3,5 and 7 are free for the user.

- 8259 Programmable Interrupt Controller

The chip responsible for handling interrupt requests in a PC is the 8259 Interrupt Controller. To use interrupts you will need to know how to read and set the 8259's internal interrupt mask register (IMR) and how to send the end-of-interrupt (EOI) command to acknowledge the 8259 interrupt controller.

-Interrupt Mask Register (IMR)

Each bit in the interrupt mask register (IMR) contains the mask status of the interrupt line. If a bit is set (equal to 1), then the corresponding IRQ is masked, and it will not generate an interrupt. If a bit is cleared (equal to 0), then the corresponding IRQ is not masked, and it can generate an interrupt. The interrupt mask register is programmed through **port 21h**.

-End-of-Interrupt (EOI) Command

After an interrupt service routine is complete, the 8259 Interrupt Controller must be acknowledged by **writing the value 20h to port 20h**.

-What exactly happens when an interrupt occurs?

Understanding the sequence of events when an interrupt is triggered is necessary to correctly write interrupt handlers. When an interrupt request line is driven high by a peripheral device (such as the ECAN527), the interrupt controller checks to see if interrupts are enabled for that IRQ, and then checks to see if other interrupts are active or requested and determines which interrupt has priority. The interrupt controller then interrupts the processor. The current code segment (CS), instruction pointer (IP), and flags are pushed onto the system stack., and a new set if CS and IP are loaded from the lowest 1024 bytes of memory.

This table is referred to as the interrupt vector table and each entry to this table is called an interrupt vector. Once the new CS and IP are loaded from the interrupt vector table, the processor starts to execute code from the new Code Segment (CS) and from the new Instruction Pointer (IP). When the interrupt routine is completed the old CS and IP are popped from the system stack and the program execution continues from the point it was interrupted.

-Using Interrupt in your Program

Adding interrupt support to your program is not as difficult as it may seem especially when programming under DOS. The following discussion will cover programming under DOS. Note, that even the smallest mistake in your interrupt program may cause the computer to hang up and will only restart after a reboot. This can be frustrating and time-consuming.

-Writing an Interrupt Service Routine (ISR)

The first step in adding interrupts to your software is to write an interrupt service routine (ISR). This is the routine that will be executed automatically each time an interrupt request occurs for the specified IRQ. An ISR is different from other subroutines or procedures. First, on entrance the processor registers must be pushed onto the stack before anything else! Second, just before exiting the routine, you must clear the interrupt on the DPM104HR and write the EOI command to the interrupt controller. Finally, when exiting the interrupt routine the processor registers must

be popped from the system stack and you must execute the IRET assembly instruction. This instruction pops the CS, IP and processor flags from the system stack. These were pushed onto the stack when entering the ISR:

Most compilers allow you to identify a function as an interrupt type and will automatically add these instructions to your ISR with one exception: most compilers do not automatically add the EOI command to the function, you must do it yourself. Other than this and a few exceptions discussed below, you can write your ISR as any code routine. It can call other functions and procedures in your program and it can access global data. If you are writing your first ISR, we recommend you stick to the basics; just something that enables you to verify you have entered the ISR and executed it successfully. For example: set a flag in your ISR and in your main program check for the flag.

Note: If you choose to write your ISR in in-line Assembly , you must push and pop registers correctly, and exit the routine with the IRET instruction instead of the RET instruction.

There are a few precautions you must consider when writing ISR's. The most important is, **do not use any DOS functions or functions that call DOS functions from an interrupt routine.** DOS is not reentrant; that is, a DOS function cannot call itself. In typical programming, this will not happen because of the way DOS is written. But what about using interrupts? Then, you could have the situation such as this in your program. If DOS function X is being executed when an interrupt occurs and the interrupt routine makes a call to the DOS function X, then function X is essentially being called while active. Such cases will cause the computer to crash. DOS does not support such operation. A general rule is , that do not call any functions that use the screen, read keyboard input and any file I/O routines should not be used in ISR's.

The same problem of reentrancy exists for many floating point emulators as well, meaning you should avoid floating point mathematical operations in your ISR.

Note, that the problem of reentrancy exists, no matter what programming language you use. Even, if you are writing your ISR in Assembly language, DOS and many floating point emulators are not reentrant. Of course, there are ways to avoid this problem, such as those which involve checking if any DOS functions are currently active when your ISR is called, but such solutions are beyond the scope of this manual.

The second major concern when writing ISR's is to make them as short as possible in term of execution time. Spending long times in interrupt service routines may mean that other important interrupts are not serviced. Also, if you spend too long in your ISR, it may be called again before you have exited. This will lead to your computer hanging up and will require a reboot.

Your ISR should have the following structure:

- Push any processor registers used in your ISR. Most C compiler do this automatically
- Put the body of your routine here
- Read interrupt status address of your DPM104HR board to clear interrupt
- Issue the EOI command to the 8259 by writing 20h to address 20h
- Pop all registers. Most C compilers do this automatically

The following C example shows what the shell of your ISR should be like:

```
/*---------------------------------------------------------------------------
| Function:     new_IRQ_handler
| Inputs:       Nothing
| Returns:      Nothing         - Sets the interrupt flag for the EVENT.
|--------------------------------------------------------------------------*/
void interrupt far new_IRQ_handler(void)
{

        IRQ_flag = 1;           // Indicate to main process interrupt has occurred
        {
        // Your program code should be here
        }
                                // Read address 1FFE or 1FFF to
                                // Clear interrupt
        outp(0x20, 0x20);       /* Acknowledge the interrupt controller. */
}
```

-Saving the Startup Interrupt Mask Register (IMR) and interrupt vector

The next step after writing the ISR is to save the startup state of the interrupt mask register (IMR) and the original interrupt vector you are using. The IMR is located in address **21h**. The interrupt vector you will be using is located in the interrupt vector table which is an array of 4-byte pointers (addresses) and it is located in the first 1024 bytes of the memory (Segment 0 offset 0). You can read this value directly, but it is a better practice to use DOS function 35h (get interrupt vector) to do this. Most C compilers have a special function available for doing this. The vectors for the hardware interrupts on the XT - bus are vectors 8-15, where IRQ0 uses vector 8 and IRQ7 uses vector 15. Thus if your DPM104HR is using IRQ5 it corresponds to vector number 13.

Before you install your ISR, temporarily mask out the IRQ you will be using. This prevents the IRQ from requesting an interrupt while you are installing and initializing your ISR. To mask the IRQ, read the current IMR at I/O port 21h, and set the bit that corresponds to your IRQ. The IMR is arranged so that bit 0 is for IRQ0 and bit 7 is for IRQ7. See the paragraph

entitled *Interrupt Mask Register (IMR*) earlier in this discussion for help in determining your IRQ's bit. After setting the bit, write the new value to I/O port 21h.

With the startup IMR saved and the interrupts temporarily disabled, you can assign the interrupt vector to point to your ISR. Again you can overwrite the appropriate entry in the vector table with a direct memory write, but this is not recommended. Instead use the DOS function 25h (Set Interrupt Vector) or, if your compiler provides it, the library routine for setting up interrupt vectors. Remember , that interrupt vector 8 corresponds to IRQ0, vector 9 for IRQ1 etc.

If you need to program the source of your interrupts, do that next. For example, if you are using transmitted or received messages as an interrupt source , program it to do that.

Finally, clear the mask bit for your IRQ in the IMR. This will enable your IRQ.

-Common Interrupt mistakes

- Remember, hardware interrupts are from 8-15, XT IRQ's are numbered 0-7
- Forgetting to clear the IRQ mask bit in the IMR
- Forgetting to send the EOI command after ISR code. Disables further interrupts.

Example on Interrupt vector table setup in C-code:

```
void far _interrupt new_IRQ1_handler(void );        /* ISR function prototype */
#define IRQ1_VECTOR       3                         /* Name for IRQ */
void (interrupt far *old_IRQ1_dispatcher)
        (es,ds,di,si,bp,sp,bx,dx,cx,ax,ip,cs,flags);    /* Variable to store old IRQ_Vector */
void far _interrupt new_IRQ1_handler(void );

/*-------------------------------------------------------------------------
| Function:     init_irq_handlers
| Inputs:       Nothing
| Returns:      Nothing
| Purpose:      Set the pointers in the interrupt table to point to
|               our funtions ie. setup for ISR's.
|-------------------------------------------------------------------------*/
void init_irq_handlers(void)
{
        _disable();
                old_IRQ1_handler = _dos_getvect(IRQ1_VECTOR + 8);
                _dos_setvect(IRQ1_VECTOR + 8, new_IRQ1_handler);
                Gi_old_mask = inp(0x21);
                outp(0x21,Gi_old_mask & ~(1 << IRQ1_VECTOR));
        _enable();
}
```

```
|/*---------------------------------------------------------------------
| Function:     restore do this before exiting program
| Inputs:       Nothing
| Returns:      Nothing
| Purpose:      Restore interrupt vector table.
|---------------------------------------------------------------------*/
void restore(void)
{
        /* Restore the old vectors */

        _disable();

                _dos_setvect(IRQ1_VECTOR + 8, old_IRQ1_handler);
                outp(0x21,Gi_old_mask);

        _enable();

}
```

APPENDIX A

==========

DPM104HR Specifications

Host Interface

Memory mapped into low memory	**16Kbytes**
Jumper-selectable base address	
16-bit data bus	
Jumper selectable interrupts XT and AT	

Connectors

Host bus (Master / Slave side)	**16-bit PC/104 busses**

Electrical

Operating temperature range	**-40 to +85C**
Supply voltage	**+5V only**
Power consumption	**1,25W**

 (c) RTD Finland Oy 1997-2001

NOTES:

(C) RTD Finland Oy 1997-2001 DOC: DPM104HR.SAM